基于 HLS 的 Sobel 算子实现与 FPGA 硬件加速的探讨

2026 年 5 月 17 日

摘要

本次实验实现了 Sobel 算子的基础卷积操作，目的是提取图像的边缘信息，为后续更复杂的图像处理方法提供支持。通过 Sobel 算子分别提取水平方向和垂直方向的边缘，最后合成为完整的边缘检测图像。使用了 HLS 工具生成 IP 核导入 Vivado 生成 Bitstream，最后使用 Jupyter 实现了图像与视频处理并且有对比 Python 处理图像的性能探讨。

本报告的结构如下：第一章引言部分介绍 FPGA、HLS 工具、Xilinx；第二章介绍 Sobel 算子的理论基础与在 FPGA 上实现 Sobel 算子卷积；第三章对测试结果进行分析；第四章对报告进行总结与未来展望；第五章是用于补充说明的附录。

注：由于课程主要探讨的是 Xilinx 开发 FPGA 的工具链，同时该图像处理是比较基础的内容（很多应用的前置操作都要使用），所以报告中不展开 Sobel 研究背景避免冗长（PPT 中会提及）。

Abstract

This experiment implemented the basic convolution operation of the Sobel operator, aiming to extract edge information from images to support more complex image processing methods in the future. The Sobel operator was used to extract horizontal and vertical edges separately and then combine them into a complete edge detection image. The HLS tool was utilized to generate an IP core, which was then imported into Vivado to generate the Bitstream. Finally, Jupyter was employed for image and video processing, along with a performance comparison between Python and FPGA implementations.

The structure of this report is as follows: Chapter 1 introduces FPGA, the HLS tool, and Xilinx; Chapter 2 explains the theoretical basis of the Sobel operator and its implementation on FPGA; Chapter 3 analyzes the test results; Chapter 4 provides a conclusion and future outlook; Chapter 5 contains appendices for supplementary information.

Note: Since the course primarily focuses on the Xilinx FPGA toolchain, and the image processing task discussed is foundational (often used as a preprocessing step in many applications), the report does not elaborate on the research background of Sobel to avoid redundancy (details will be covered in the presentation slides).

目录

Chapter 1

引言

1.1 FPGA

本门课程基于的硬件平台是 FPGA（Field-Programmable Gate Array），所以引言部分还是很有必要大致的讲解一下 FPGA 相关的前置知识。正如它的名字一样，FPGA 是现场可编程门电路，是一种介于专用处理器（ASIC）与通用处理器之间的特殊硬件。通用处理器如 CPU、GPU 等因为要适配运行各种各样的软件，有一套复杂的指令集，带来的后果就是在某些具体领域的使用上可能不如专用处理器（ASIC），但是 ASIC 存在不够灵活和开发周期长的缺点，所以 FPGA 这种“半定制”的硬件产品诞生了。人们期望他有比通用处理器在某个具体领域有更好的性能，同时又希望拥有一定的灵活性和较快开发周期，最好能做到与 ASIC 性能差距不要太大。

在更进一步讨论 FPGA 之前很有必要介绍一下 Jonathan Rose 与他的部分重要研究成果。首先 J.Rose 提出了最基本的 FPGA 架构，包含两部分，分别是逻辑块（Logic Block）与路由架构（Routing Architectures）. 并且系统性地研究了这两部分对 FPGA 的逻辑密度（Logic Density）和性能的影响，尤其是通过对彼时 Xilinx 商用 FPGA 中逻辑块设计的分析，揭示了逻辑块粒度（Granularity）与设计效率之间的权衡关系 [1]。同时他在研究 FPGA 架构时提出了逻辑块功能性与面积效率之间的权衡关系，指出最优的 LUT 输入数量为 3 到 4，并强调包含触发器的逻辑块设计在面积效率上的重要性。同时，他们的研究首次揭示了路由面积在 FPGA 总面积中的

主导作用 [2]。总的来说，以 J. Rose 为代表的研究者通过深入分析 FPGA 的逻辑块与路由架构，奠定了现代 FPGA 的基础。其在逻辑块功能性与面积效率优化方面的开创性工作，为后续 FPGA 架构设计的持续发展提供了理论支持。时至今日，FPGA 的功能已从当初的简单门阵列发展为支持高性能计算、人工智能加速等多领域应用的多功能平台，这离不开这一系列研究的推动和积累。

在 2006 年就有关于 FPGA 对比 ASIC 的讨论，彼时的结论是 FPGA 平均面积是 ASIC 的 40 倍，同时速度慢 3.2 倍，动态功耗高 12 倍 [3]。虽然在当时看起来 FPGA 相较于 ASCI 性能差距太大。但是在 2008 年 Ian Kuon 等人就指出 FPGA 的灵活性和快速开发能力使其在低产量和快速迭代应用中更具吸引力。随着工艺节点的缩小，FPGA 在速度和功耗方面的差距也逐步缩小 [4]。16 年后的今天，随着 FPGA 工艺与设计的发展，FPGA 与 ASIC 的差距也在逐渐的拉近。同时在对比通用处理器的时候，FPGA 更大的优势是低功耗应用，能耗比相较于通用处理器（特别是 GPU）的”力大砖飞“来说好很多。这里以相关研究团队的论文作为例子进行探讨。该篇论文使用 XC7VX980T 实现了 VGG-16（visual geometry group network-16）算法。虽然论文中没有提到具体的功耗，但是在一系列的优化后实现了在 150MHz 时钟频率下的 1TOPS 算力，同时达到了理论吞吐量的 98.15% [5]。这说明了 FPGA 在具体某个领域的应用有非常大的优势。

FPGA 作为一种灵活的硬件平台，在多个高性能计算和嵌入式系统领域中展现出了广阔的发展潜力。特别是在激光雷达领域，FPGA 已经成为许多企业实现核心控制和数据处理的首选平台。例如，国内领先的激光雷达厂商速腾聚创（RoboSense）和禾赛科技（Hesai Technology）均在其产品中采用 FPGA 作为主控平台。FPGA 的高并行计算能力和可编程性使其能够高效地满足激光雷达在实时信号处理、点云生成和数据传输等方面的高性能需求。

以速腾聚创为例，其通过基于 FPGA 平台的版本迭代策略，成功在统一的硬件架构基础上实现了高端、中端和低端激光雷达产品的差异化开发。这种方法不仅优化了产品性能，还显著降低了产品开发的时间与研发成本，为企业提供了更高的市场竞争力。

总的来说，FPGA 在激光雷达领域的应用不仅提升了产品性能和开发

效率，也为未来技术创新和产业升级提供了广阔的空间。这种技术路径的成功案例表明，FPGA 作为一种强大的可编程硬件平台，在多个前沿领域中具有重要的应用价值和发展潜力。

1.2 HLS 工具

高层次综合（High-Level Synthesis, HLS）是一种将高抽象级别的程序（如 C、C++ 或 SystemC 描述）自动转换为硬件描述语言（HDL，例如 Verilog 或 VHDL）的技术。HLS 的核心目标是通过提升设计抽象层次，将设计者从传统 RTL（寄存器传输级）设计的复杂性中解放出来，使其能够更专注于算法和系统行为的开发。HLS 工具的基本流程包括从高层次描述中自动完成硬件资源分配、操作调度、变量绑定，并最终生成优化的硬件架构（如数据通路和控制器）。通过 HLS，开发者能够高效探索设计空间，优化性能、面积和功耗，从而加速硬件开发和验证过程 [6]。

一家开发 HLS 的初创公司 AutoESL Design Technologies Inc 在 2011 年被 Xilinx 收购，这是一个非常重要的里程碑事件，自此之后，全世界最大的 FPGA 开发商开始进军 HLS 工具的开发。HLS 发展到现在十多年在一些领域也确实有比较成功的地方，比如深度学习（Deep Learning）、视频转码（Video Transcoding）、图处理（Graph Processing）、基因组测序的加速（Acceleration of Genome Sequencing）等。与此同时 HLS 的发展面临诸多挑战，例如如何进一步简化 FPGA 编程流程，降低软件开发人员的使用门槛；如何高效移植传统代码以适配 FPGA 的并行计算架构；如何推动更多开源项目的出现以完善生态；以及是否需要设计专注于 HLS 的新型编程语言，以更好支持硬件特性的表达和优化。这些问题的解决将直接影响 HLS 技术的普及与应用前景 [7]。

综上所述，高层次综合（HLS）通过提升硬件设计的抽象层次，极大地降低了硬件开发的复杂性，并在深度学习、视频处理、图处理和基因组测序等领域展现了显著的优势。然而，HLS 的进一步发展仍需克服编程简化、传统代码移植、开源生态建设以及专用语言设计等多方面的挑战。只有在技术和生态持续完善的基础上，HLS 才能更广泛地推动 FPGA 的普及与创新应用，为更多领域的硬件加速提供高效解决方案。

1.3 Xilinx

Xilinx 作为 FPGA、可编程 SoC 和 ACAP（自适应计算加速平台）的发明者，其技术创新和市场领导地位都毋庸置疑。从市场占有率到技术高度，Xilinx 始终是 FPGA 领域的领军企业，其产品和技术推动了可编程硬件的发展，并广泛应用于通信、数据中心、人工智能、医疗、汽车等行业。因此，详细了解 Xilinx 的发展历程及其在 HLS 和 FPGA 生态中的关键作用，对于全面认识 HLS 技术的发展背景和未来趋势显得尤为重要。

Xilinx 创建于 1984 年，总部位于美国加利福尼亚州硅谷的圣荷西，是 FPGA、可编程 SoC 和 ACAP 的发明者，同时也是全球 FPGA 市场的领导者。多年来，Xilinx 通过一系列战略性收购和技术创新不断扩展其产品线和市场影响力。2011 年，Xilinx 收购 AutoESL Design Technologies Inc，开始进军 HLS 工具发展。2018 年，Xilinx 收购了中国人工智能芯片初创公司深鉴科技，进一步强化其在 AI 和深度学习领域的布局；2019 年收购了 Solarflare Communications，以增强其高性能网络能力。2020 年，Xilinx 宣布以换股方式被美国芯片制造商 AMD 以 350 亿美元估值收购，并于 2022 年 2 月完成交易，自此成为 AMD 自适应和嵌入式运算事业部的一部分。与此同时，Xilinx 持续推动创新，推出了 Vitis 平台、Kria 系列小型系统模块（SOM），并通过收购 Falcon Computing Systems 和 Silexica 等公司进一步增强其软件与生态系统能力。如今，Xilinx 的技术与产品已全面融入 AMD 品牌，继续推动可编程硬件技术的发展。

在开发工具链上，即使作为领跑者的头部厂商，Xilinx 也并未松懈，不断的推出新的工具链以及新的开发方法。从近些年的不断整合平台可以看出 Xilinx 确实是想要将自己打造一个 All in One 的一个强大开发生态。2020 年，Xilinx 发布了 Vitis，这是一款功能强大的集成开发平台，旨在统一硬件和软件开发环境，显著提升开发效率和灵活性。Vitis 平台覆盖了从 HLS 到 ARM 编程的完整软件开发流程，结合其对异构计算架构的支持，为开发者提供了强大的工具链和生态支持 [8]。随着版本的迭代更新，Vitis 的功能进一步完善，例如在最新的 24.2 版本中，HLS 的 GUI 已完全融入 Vitis 平台中，标志着其开发模式的全面整合。未来，Xilinx 将围绕 Vitis 和 Vivado 平台进行功能划分，明确由 Vitis 负责 HLS、ARM 编程等软件开发功能，而 Vivado 则专注于 FPGA PL 端的硬件开发与优化，形成软硬件分工协作

的完整开发生态体系。

总的来说，Xilinx 目前在技术能力和市场地位上仍然处于行业领先，其工具链和生态系统在 FPGA 领域中具有不可替代的地位。对于每一位学习和开发 FPGA 的人来说，掌握 Xilinx 的相关工具和技术已经成为不可避免的重要环节。

Chapter 2

Sobel 算子的梯度算法理论分析与实现

2.1 Sobel 算子理论

Sobel 算子是在计算机图像处理领域中被广泛应用。在详细讲解 Sobel 算子处理图像的原理和步骤之前，有必要对计算机图像的基本概念进行简要的介绍，以便更好地理解其应用背景。

以 RGB 图像为例，数字图像由三个通道（红色 R、绿色 G 和蓝色 B）组成，每个通道中的像素值表示该通道的颜色强度。像素值的取值范围取决于图像的位深，例如，常见的 8 位图像，其像素值范围为 0 至 255，表示从颜色强度最低到最高的变化（图 2.1）。然而，对于 Sobel 算子而言，其关注的重点是图像中像素强度的梯度变化，而非具体的颜色信息。因此，在应用 Sobel 算子之前，通常需要将 RGB 图像转换为灰度图像 [9]，使其更加适合梯度计算的需求。

RGB 转换为灰度图的过程基于加权叠加的方法，即对图像的三个通道的像素值按照一定的权重系数进行加权平均，以生成单一数值来表示“灰度”。这一加权计算的常用标准来源于 ITU-R BT.601 色彩编码标准，其中对红、绿、蓝三通道分别赋予权重系数 0.299、0.587 和 0.114（公式 2.1）。这样的分配是基于人眼对不同颜色敏感度的差异：人眼对绿色最为敏感，其次是红色，对蓝色的敏感度最低。因此，该公式更贴合人类视觉感知的亮度

效果，能够有效地保留图像的亮度信息。

最终我们的得到了代表图像亮度的灰度图，在进行 Sobel 算子的讲解之前，很有必要讲解一下什么是卷积操作。简单来说就是用一个比需要卷积的图像更小的矩阵（称之为卷积核）对图像进行一个轮询矩阵相乘，得到的数值保存为卷积的结果，形成最后的卷积后矩阵（图 2.2）。

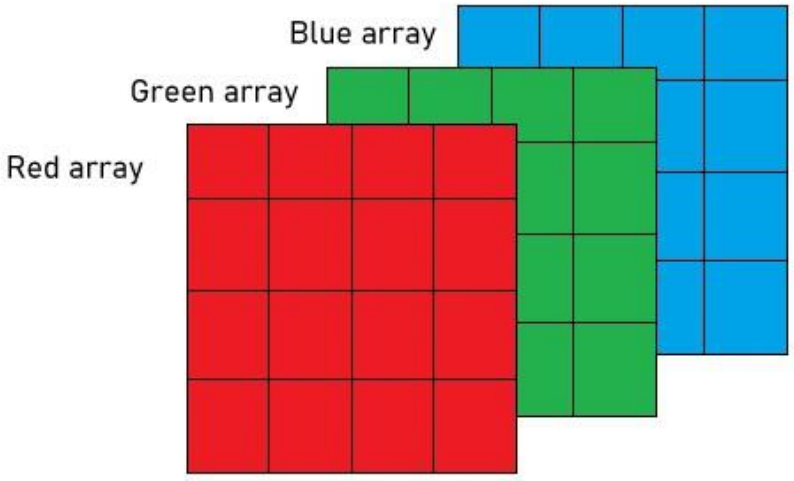


图 2.1: 基本的 RGB 图像组成

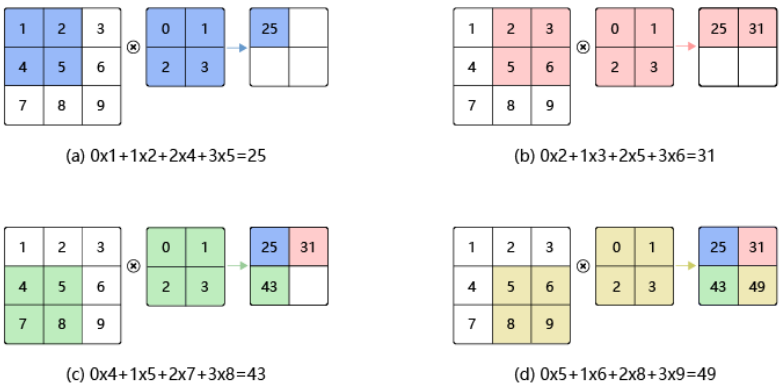


图 2.2: 基本的卷积操作

$$Gray = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{2.1}$$

在明白了什么是卷积后，我们可以进一步了解 Sobel 算子。Sobel 算子卷积的目的是得到图像的梯度信息，为了能更好的理解 Sobel 卷积的原理，我们从二阶算子讲起。一个二阶 sobel 算子如下所示：

$$\Delta x = \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}, \quad \Delta y = \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix}$$

假设我们有一个矩阵是三阶矩阵，可以发现我们使用 Δx 与 Δy 进行卷积的结果为：

$$\text{输入矩阵} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad \Delta x \text{ 卷积结果} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \Delta y \text{ 卷积结果} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

这反映了在 x 方向上的梯度有所变化，y 方向的梯度没有变化。通过这一示例可以清楚地理解二阶 Sobel 算子的工作原理。而在实际工程应用和本次实验中，常用的三阶 Sobel 算子（即更大尺寸的卷积核）也基于相同的原理，适用于更复杂的场景下的梯度计算。

三阶 Sobel 算子的卷积核如下所示：

$$\Delta x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \Delta y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

我们使用三阶 Sobel 算子进行图像的卷积即可得到某个方向的梯度（图 2.3），最后使用梯度公式将两个方向的梯度幅值相加（公式 2.2）使用 Python 代码在 FPGA 上可以简单的实现（如何搭建 PYNQ-Z2 的 Jupyter Notebook 环境参见附录 5.3），实现的代码和结果如图 2.4、图 2.5。

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.2}$$

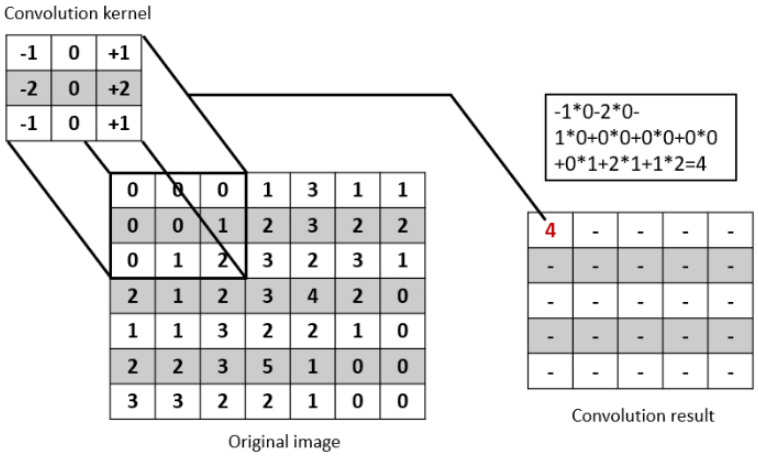


图 2.3: 使用三阶 Sobel 算子进行卷积

```python
import time
start_time = time.time()
sobel_x =cv2.Sobel(gray,cv2.CV_8U ,1,0)
sobel_y = cv2.Sobel(gray,cv2.CV_8U ,0,1)
sobel_res = np.clip(sobel_x + sobel_y, 0, 255)
end_time = time.time()
print("Time cost with Python: {}s".format(end_time - start_time))
fig_sobel3 = plt.figure()
fig_sobel3.set_figheight(4)
fig_sobel3.set_figwidth(15)

# gradient x
fig_1 = fig_sobel3.add_subplot(131)
fig_1.title.set_text('Gradient X')
plt.imshow(sobel_x,cmap='gray')

# gradient y
fig_2 = fig_sobel3.add_subplot(132)
fig_2.title.set_text('Gradient Y')
plt.imshow(sobel_y,cmap='gray')

# gradient
fig_3 = fig_sobel3.add_subplot(133)
fig_3.title.set_text('Gradient X + Y')
plt.imshow(sobel_res,cmap='gray')
```

图 2.4: Python 实现 Sobel 算子卷积

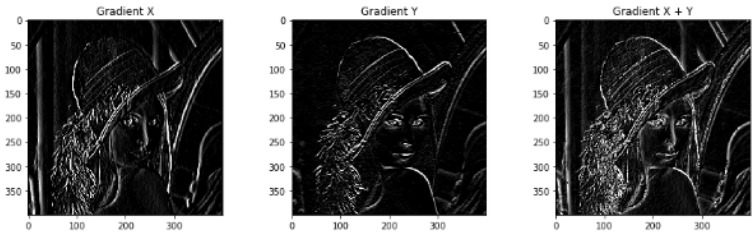


图 2.5: 卷积后 X 和 Y 方向的图像与叠加后的图像

2.2 HLS 生成 IP 核

在工程开发中，我们首要关注的是实现然后才是优化。本小节专注于如何使用 HLS 实现 Sobel 卷积。我们只需要注意乘 2 使用左移是一个比较快捷高效的操作。同时忽略掉乘以 0 的部分。最终得到的 X 和 Y 方向的卷积核操作（图 2.6）。

接下来我们要考虑的是如何合并 X 和 Y 方向的卷积核操作。首先我们简化了之前提到的梯度计算公式 (公式 2.2)，使用了近似公式（公式 2.3）。叠加后的操作就是两个卷积核的相加（图 2.7），实现代码就是调用前面定义的函数把函数结果相加（图 2.8）

$$G = G_x + G_y \tag{2.3}$$

在完成一个像素的卷积核的实现后，需要进一步设计整个图像的遍历方法，以便依次向卷积核提供包含当前像素及其周围 8 个相邻像素的窗口数据。一个直观且有效的方法是忽略图像边缘区域，从而限定卷积核中心可遍历的有效区域，即图像中以深红色框表示的矩形区域（图 2.9）。随后，通过“滑动”卷积核的中心位置（图 2.9中的黄色区域），逐一遍历整个图像的有效区域。在每一次移动中，卷积核会覆盖新的 3x3 区域，并从存储器中读取覆盖区域的 9 个像素值。这些像素值将被传递至卷积核函数，完成当前中心像素的梯度计算。

于是我们能得到一个最基础版本的 Sobel 卷积操作（图 2.10），也就是按照每一次读取 9 个像素点的方法，重复调用卷积的函数实现。这个函数从逻辑上没有问题，也能实现我们所需要的效果。

然而，当考虑在 FPGA 上实现时，这种基础实现存在较大的优化空间。

```
8   static PIXEL Gradient_X(PIXEL WB[3][3])
9   {
10      short int M00 = ((short int)WB[1][0] << 1);
11      short int M01 = ((short int)WB[1][2] << 1);
12      short int A00 = (WB[0][2] + WB[2][2]);
13      short int S00 = (WB[0][0] + WB[2][0]);
14      short int out_pix;
15      out_pix = M01 - M00;
16      out_pix = out_pix + A00;
17      out_pix = out_pix - S00;
18
19      if(out_pix<0)
20      {
21          out_pix = 0;
22      }
23
24      if(out_pix>255)
25      {
26          out_pix = 255;
27      }
28      return (PIXEL) out_pix;
29  }
30
31  static PIXEL Gradient_Y(PIXEL WB[3][3])
32  {
33      short int M00 = ((short int)WB[0][1] << 1);
34      short int M01 = ((short int)WB[2][1] << 1);
35      short int A00 = (WB[2][0] + WB[2][2]);
36      short int S00 = (WB[0][0] + WB[0][2]);
37      short int out_pix;
38      out_pix = M01 - M00;
39      out_pix = out_pix + A00;
40      out_pix = out_pix - S00;
41
42      if(out_pix<0)
43      {
44          out_pix = 0;
45      }
46      if(out_pix > 255)
47      {
48          out_pix = 255;
49      }
50      return (PIXEL) out_pix;
51  }
```

图 2.6: C++ 实现 X 与 Y 方向的 Sobel 算子卷积

由于 FPGA 的并行计算特性和资源使用的特点，这种逐像素的卷积方法并未充分发挥硬件的性能优势。在接下来的小节中，我们将对这一基础实现进行分析，并探讨如何通过优化设计提升其在 FPGA 上的执行效率。

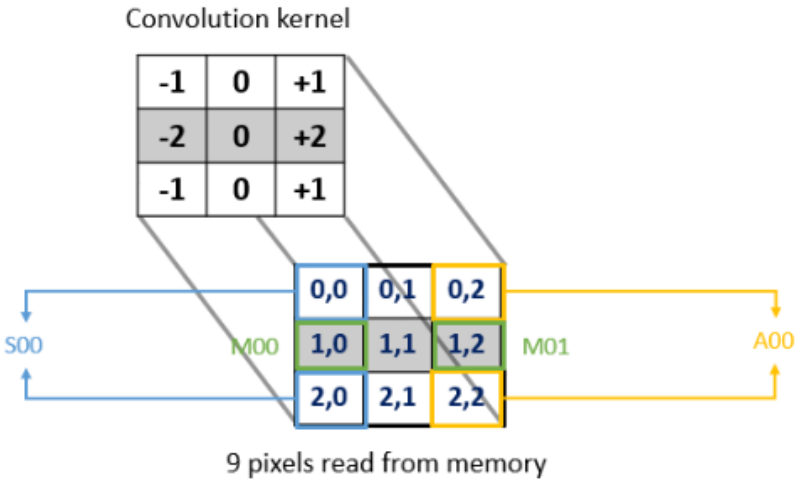


图 2.7: 可视化 Sobel 算子 X 与 Y 方向叠加

```
48  static PIXEL sobel3x3_kernel(PIXEL WB[3][3])
49  {
50      PIXEL g_x, g_y, sobel;
51      short temp;
52      g_x = Gradient_X(WB);
53      g_y = Gradient_Y(WB);
54      temp = g_x + g_y;
55      if(temp > 255) sobel = 255;
56      else sobel = temp;
57
58      return sobel;
59  }
```

图 2.8: 叠加 XY 方向的 Sobel 算子函数

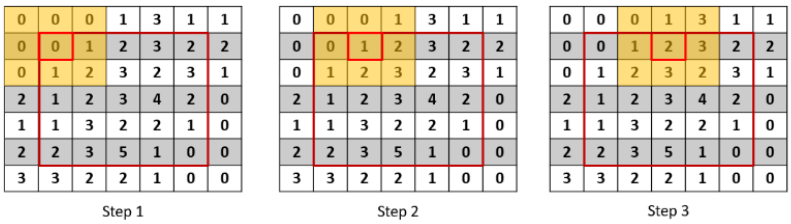


图 2.9: 基本的滑动 Sobel 卷积操作

```cpp
void naive_sobel(PIXEL* src, PIXEL* dst, int rows, int cols)
{
    int row, col;
    PIXEL sobel_kernel[3][3];
    for(row = 0; row < rows+1; row++)
    {
#pragma HLS LOOP_TRIPCOUNT min=1 max=720
        for(col = 0; col < cols+1; col++)
        {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1280
            PIXEL _sobel;

            if(row<=1 || col<=1 || row>(rows-1) || col>(cols-1))
                _sobel = 0;
            else
            {
                for(int i=0; i<3; i++)
                {
                    for(int j=0; j<3; j++)
                    {
                        sobel_kernel[i][j] = src[(row+i-1)*cols+(col+j-1)];
                    }
                }
                _sobel = sobel3x3_kernel(WB: sobel_kernel);
            }
            if(row>1 && col>1)
            dst[(row-1)*cols+(col-1)] = _sobel;
        }
    }
}
```

图 2.10: 基本的滑动 Sobel 卷积操作 C++ 实现

2.3 IP 核优化

使用最基础版本的卷积代码进行综合得到的结果并不好（图 2.11）。在进行讨论前有必要说明一下综合后结果中的几个重要参数。首先 TRIP COUNT 是 921600 是由测试图片是 1280 乘 720 像素得到的，也就是 Sobel 卷积总共要处理的像素点。INTERVAL 指的是两次处理之间间隔的时钟周期。LATENCY(CYCLES) 指的是完成 Soble 函数所需要的周期。LATENCY(NS) 指的是完成 Sobel 函数所需要的时间。针对我们的卷积代码，可以看到总耗时是 82.94ms，其中比较严重的问题之一是每一次移动卷积核都要等待 9 个周期用于读取 9 个像素点。

我们一步步考虑优化，首要的问题是读取像素点。所有在红色滑动窗口内的像素我们都要读取 9 个像素点，也就是需要 1278*718*9 = 8258K 像素点。但实际上图像整个像素点才 921K 个点这是一种非常严重的浪费。实际上稍加思考我们就能发现存在很多的不必要重复读取。以 X 方向的卷积为例子（Y 方向同理）我们从红色窗口移动到蓝色窗口的时候，显然只有三个像素点进行了变更，我们却读取了整整 9 个像素点，所以可以考虑像素点数据的复用（图 2.12）。

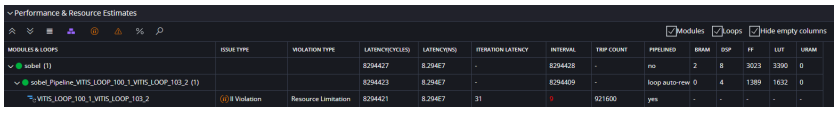


图 2.11: 未优化的综合结果

在成功考虑了每一次移动把读取 9 个像素点降低至读取 3 个像素点后，我们可以考虑是否存在更优的方法能更进一步降低这个读取的需求。在硬件设计中，可以借鉴 CPU 设计中的一个经典理念——Cache（缓存）。Cache 是现代 CPU 中非常重要的组成部分，其主要目的是通过多级缓存机制加速数据的访问速度。现代 CPU 通常设计有 L1、L2、甚至 L3 缓存，利用这些缓存来显著减少从主存（内存）读取数据的延迟。其基本原理是通过预测程序可能访问的数据，将数据提前加载到缓存中，从而在需要时快速访问 [10]。同样的思想可以应用于图像处理中的卷积操作。在卷积中，我们需要反复访问窗口中覆盖的像素值。为了提高数据访问效率，可以将这些窗口像素值视为一种特殊的缓存，称为 Window Buffer。这种缓存机制减少了

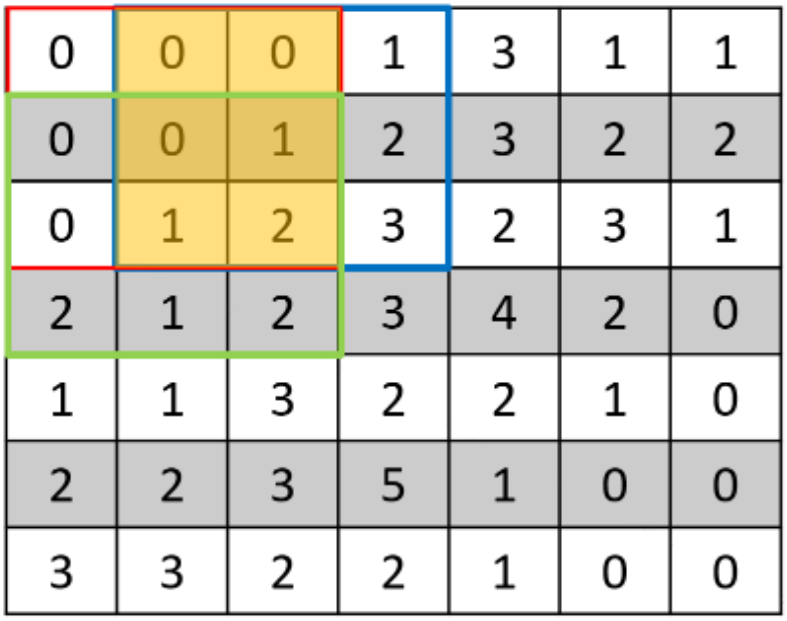


图 2.12: 可以优化的输入像素点

对全局存储器的访问，极大提高了硬件的处理效率。

具体到卷积操作，由于卷积是逐行进行的，为了进一步优化，可以设计为一次缓存图像的三行数据（即卷积核的高度），并将这三行存储到一个缓冲区中，称为 Line Buffer。Line Buffer 的设计不仅可以缓存当前正在处理的行，还可以保留上一行和下一行的数据，从而在卷积计算时直接从缓冲区中读取像素值，而无需每次从全局存储器中加载。这种方法与 CPU Cache 的设计理念类似，能够有效降低数据访问延迟，并提高硬件的吞吐量。通过结合 Window Buffer 和 Line Buffer 的机制，我们可以在硬件层面上实现类似 CPU Cache 的高效数据复用策略，从而提升卷积操作的整体性能和能效（图 2.13）。

接下来我们探讨使用 Window Buffer 与 Line Buffer 进行滑动窗口。可以看到我们直接预先加载了两行加两个像素点到 Line Buffer 里。这是 Window Buffer 开始进行读取，第一个窗口读取一个像素值，移动后的第二个窗口只需要读取一个新的值（Line Buffer 里数据可以高速传输），以此类推直到完成整个行的卷积。此时一直读取的第三行就形成了新的 Line

Buffer 里的值，此时换行读取的时候只需要复用上一行读取的 Buffer，也是每次读取一个像素点。重复操作直到完成整个图像的 Sobel 卷积（图 2.14、图 2.15）。

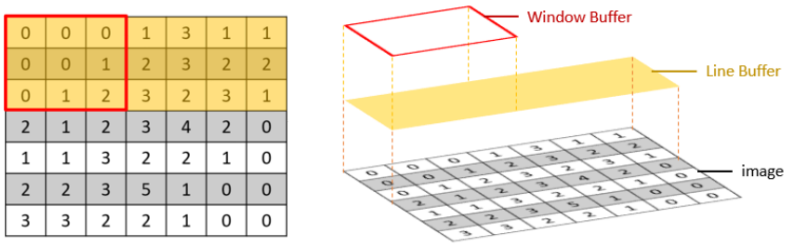


图 2.13: Window Buffer + Line Buffer

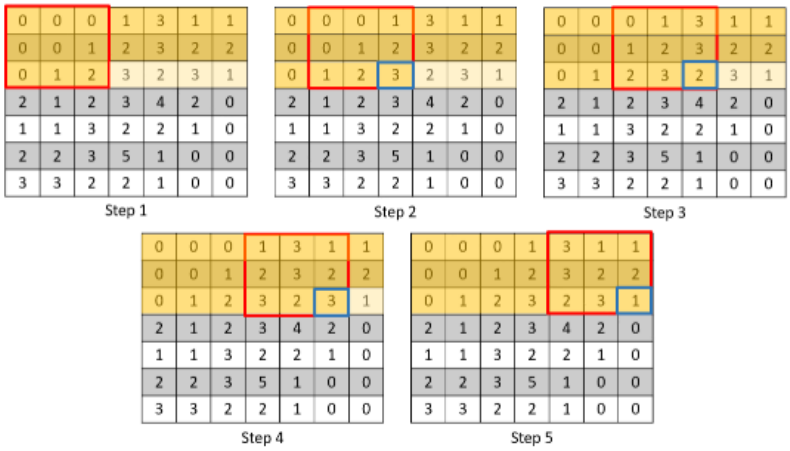


图 2.14: 使用 Buffer 后的读取单像素点滑动操作

可以看到我们使用了一系列的优化后，每次移动读取 9 个像素值变成了读取 1 个像素值。从代码角度我们将数组分割成两个高速缓存进行读取，同时我们把读取一整行的操作也流水线化，最终得到了最终版本的优化后代码（代码较长，放在附录 5.2）。此时可以看到综合后的每个循环间隔的周期下降为了 1，同时整个 Sobel 操作的时间变成了 9.2ms，相较于之前提升了 9 倍（图 2.16）!

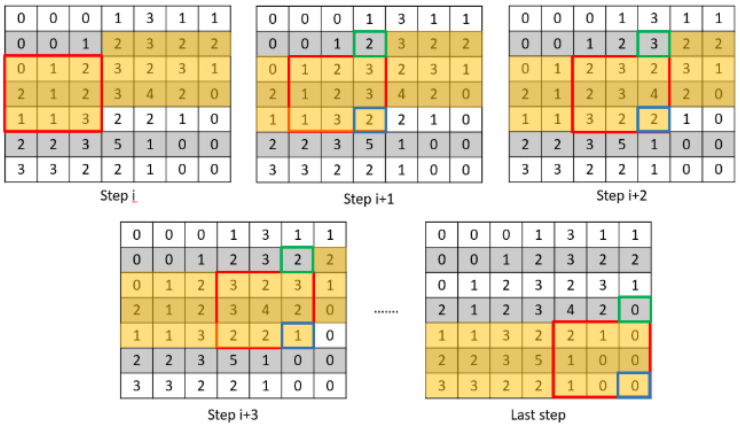


图 2.15: 使用 Buffer 后的读取单像素点滑动操作-2

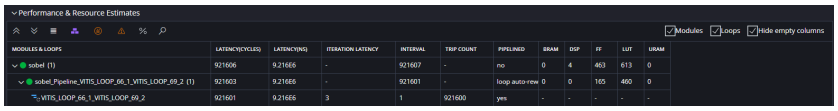


图 2.16: 优化后的综合结果

2.4 FPGA 硬件实现

在完成了 IP 核的设计后，下一步很重要的是在 Vivado 中实现并生成 Bitstream。本次实验的 Block Design 包含 Zynq Processing System、AXI Direct Memory Access、Sobel IP 核、AXI Interconnect、Reset Controller（图 2.17）。下面将一一介绍他们的作用。首先最核心的 Zynq Processing System 是 FPGA 的 Processing System，实现 Jupyter 运行 Python 与通过 DMA 通路通信硬件的功能。AXI Direct Memory Access 部分负责在 PS 的 DDR 内存和 PL 的 Sobel IP 核之间传输数据，可以实现大分辨率图像的数据流传输，减少 CPU 的数据搬运压力。Sobel IP 核是上一小节实现的 Sobel 卷积的 IP 核心，输入图像输出卷积后的图像。AXI Interconnect 实现高速数据流从 PS 到 PL 再回到 PS ，提供数据传输路径，保证 AXI 总线的读写操作能够正确路由。Reset Controller 提供全局复位信号，控制整个硬件设计的复位逻辑，在上电或需要复位时，确保所有模块能够按照正确的顺序初始化。

将 IP Block 设计完成后，就可以进行综合和实现（图 2.18）。实现后导出.bit 文件与.hwh 文件。.bit 文件的作用是配置 FPGA 的硬件功能，.hwh 作用是解析硬件的结构和端口用于调用具体寄存器等。导出后我们使用 PYNQ-Z2 的 Jupyter 环境进行调用与测试（如何配置 Jupyter 与测试代码见附录 5.3）。总体来说，本章完成了使用 Xilinx 工具链实现从高层次语言到硬件描述语言，再回到高层次语言的完整流程（图 2.19）。

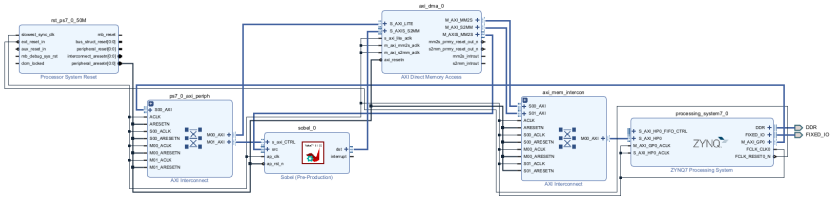


图 2.17: Vivado Block Design

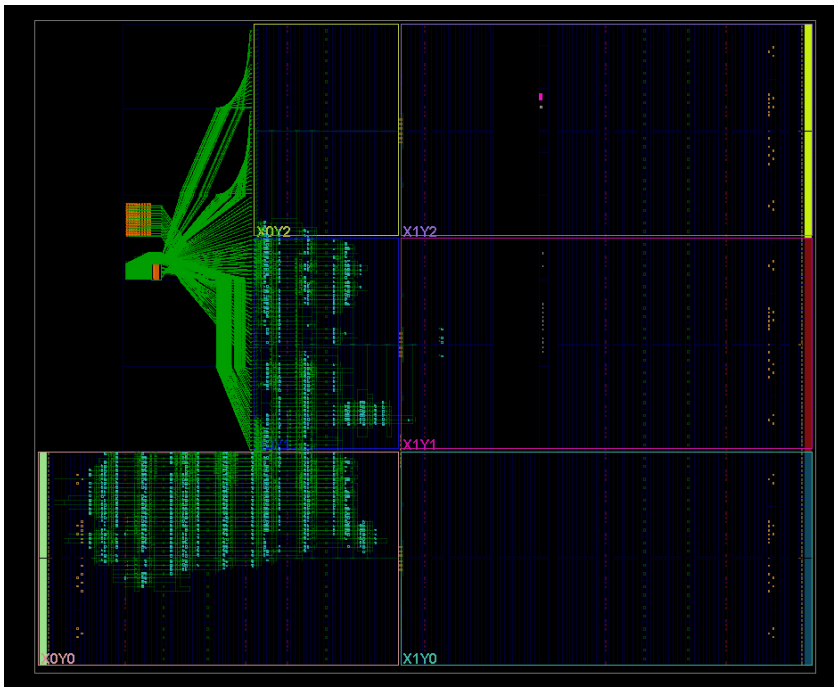


图 2.18: 综合实现后的电路图

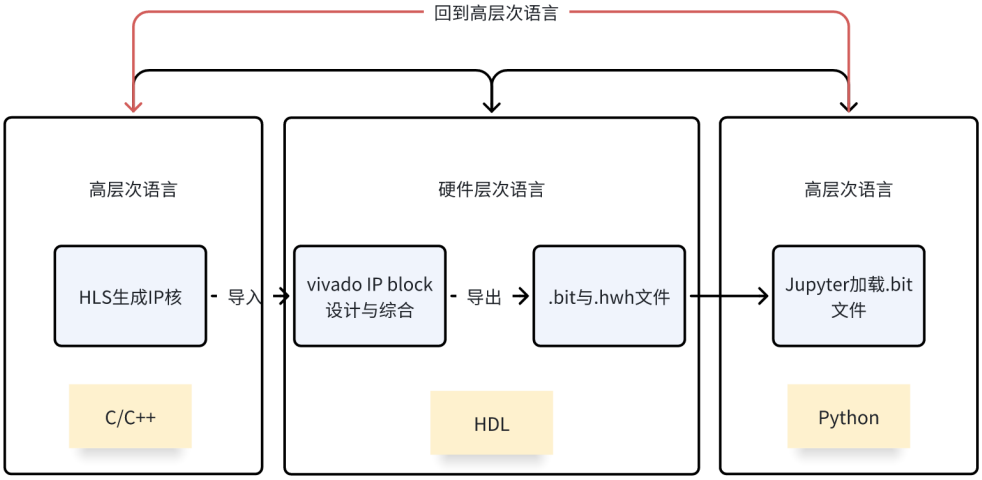

回到高层次语言
高层次语言
HLS生成IP核
导入
C/C++
硬件层次语言
vivado IP block
设计与综合
导出
.bit与.hwh文件
HDL
高层次语言
Jupyter加载.bit
文件
Python

图 2.19: 总体流程图

Chapter 3

测试结果分析

在本次实验中，我们选用了从 2048×2048 分辨率到 128×128 分辨率的五种测试图像作为测试数据集。每种分辨率之间按比例缩小，每次缩小一半，最终形成 5 种分辨率的测试图像。通过测试结果可以观察到，随着图像分辨率的降低，IP 核相较于 Python 软件实现的性能优势逐渐减小。这一现象与我们的设计预期相符，这是因为当图像分辨率较大时，IP 核的高效并行计算能力得到了充分发挥。而随着图像分辨率的减小，并行能力所带来的优势体现逐步下降，同时，由于图像数据量较小，数据读取与搬运等基础操作的时间占据了更大的比重，从而显著削弱了计算性能的提升（表 3.1）。

在实际测试中，我进一步尝试了分辨率从 64×64 到 16×16 的极小图像尺寸。测试结果表明，在这些较低分辨率下，IP 核的计算耗时几乎不再进一步降低，同时，由于测试环境中计时误差的存在，浮动幅度达到了约 1 毫秒的水平。这表明在较低分辨率下，IP 核的计算性能已经接近理论最短时间，进一步缩小分辨率的测试已无显著意义（图 3.1）。

值得注意的是，在 128×128 分辨率的测试中，观察到 IP 核相较于 Python 提升倍率出现了轻微回升。这一现象可以归因于测试中的抖动效应（如硬件时钟、DMA 操作的不稳定性、Arm 核运行 Python 不稳定等），而非实际性能的变化。因此，从整体趋势来看，图像分辨率的降低确实会导致 IP 核与 Python 软件实现之间的性能优势逐渐减小。实验结果表明，对于高分辨率图像，IP 核在硬件并行能力的加持下表现出显著的计算加速优势，而在低分辨率下，由于基础操作耗时的增加，其优势会逐步被削弱（图 3.2）。

这一实验结果为我们理解硬件加速的适用场景提供了重要的指导，表明 IP 核的性能优势在较大数据规模下最为显著，而在小数据规模下，性能优化的潜力受限于其他的系统瓶颈。（所有 Python 与 IP 核处理的图像在附录 5.4）

Resolution	Python Time (ms)	IP Core Time (ms)	Time Ratio (Python/IP)
2048	1337.3	138.6	9.65
1024	341.6	37.7	9.06
512	87.5	11.6	7.54
256	23.4	5.2	4.5
128	17.1	3.5	4.89

表 3.1: Python 和 FPGA IP 核的执行时间对比

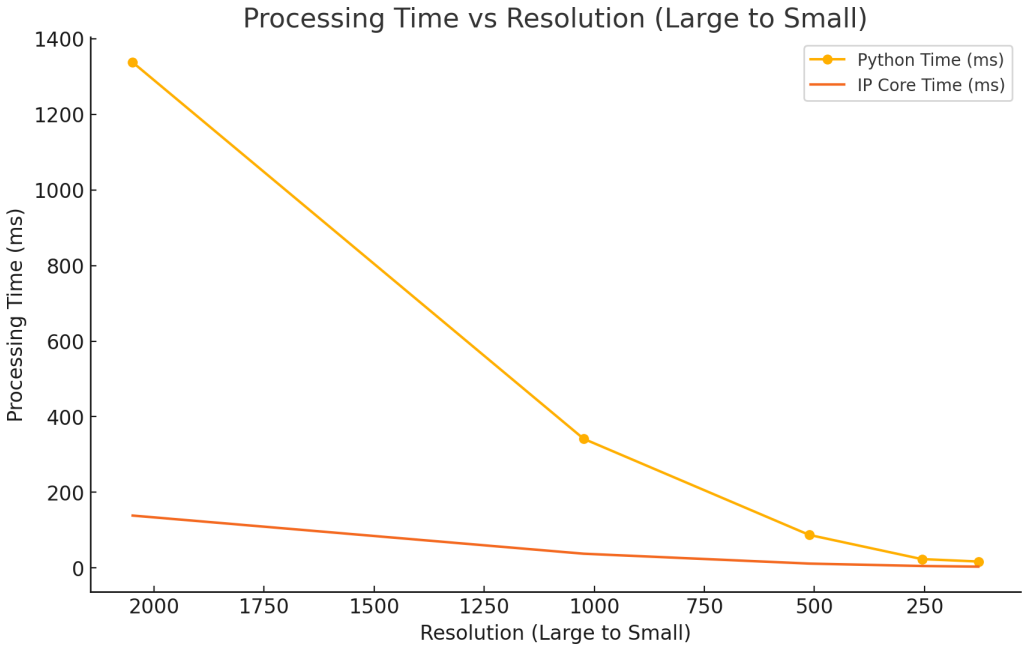


图 3.1: 处理时间对比

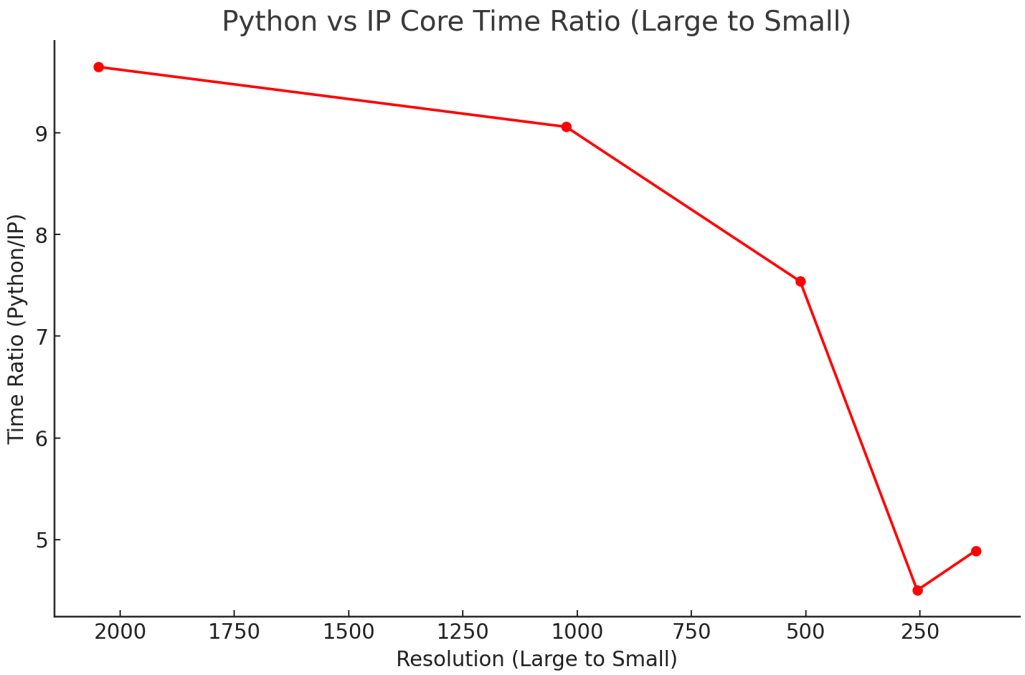


图 3.2: 处理时间倍率对比

Chapter 4

总结与未来展望

4.1 总结

本次实验以 FPGA 平台为基础，全面探讨了 Sobel 算子的实现与优化。从理论分析到实践测试，实验展示了 FPGA 在边缘检测任务中的高效性和灵活性。FPGA 作为一种半定制硬件，凭借高并行计算能力和低功耗优势，在深度学习、图像处理等领域表现出显著潜力。本实验通过高层次综合（HLS）工具生成 Sobel 算子 IP 核，并针对初步实现中的数据冗余问题进行了优化。通过借鉴现代 CPU 的缓存设计思想，引入了 Window Buffer 和 Line Buffer 机制，显著提高了卷积操作的效率。

实验结果表明，FPGA 的硬件加速在高分辨率图像处理上具有显著优势，充分发挥了其并行计算能力；而在低分辨率场景中，由于数据传输等基础操作的限制，加速效果有所减弱。本实验验证了 FPGA 平台在边缘检测任务中的有效性，为后续基于 FPGA 的硬件加速研究和应用提供了理论支持与实践经验，同时为未来课程学习奠定了良好基础。

4.2 未来展望

本次课程报告中实现的内容仅仅是图像处理领域中一个非常基础且局限的部分——Sobel 算子的实现与优化。虽然本次实验完成了从 HLS 生成 IP 核到 FPGA 硬件加速的完整流程，但这一工作在整个图像处理和硬件加

速领域中只能算是一个起点。随着 Xilinx 工具链的持续更新，尤其是 Vitis 和 Vivado 平台功能的逐步完善，未来可以基于本次实验的内容，探索更多功能和优化方向，进一步拓展其应用场景。比如可以基于 soble 算子做边缘检测算法 [11]，并将检测结果应用于目标识别、形状分析等任务。此外，结合 FPGA 的实时处理能力，还可以尝试将现有设计扩展到实时视频流的处理，通过硬件加速实现高分辨率视频中的边缘检测任务。与此同时，针对更复杂应用的硬件设计也可以进一步优化。

参考文献

[1] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1217–1225, Oct. 1990, tLDR: It was observed that the area efficiency of a logic block depends not only on its functionality but also on the average number of pins connected per logic block. [Online]. Available: http://ieeexplore.ieee.org/document/62145/

[2] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1013–1029, Jul. 1993, tLDR: A survey of field-programmable gate array (FPGA) architectures and the programming technologies used to customize them is presented and a classification of logic blocks based on their granularity is proposed, and several logic blocks used in commercially available FPGAs are described. [Online]. Available: http://ieeexplore.ieee.org/document/231340/

[3] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*. Monterey California USA: ACM, Feb. 2006, pp. 21–30, tLDR: Experimental measurements of the differences between a 90- nm CMOS field programmable gate array (FPGA) and 90-nm CMOS standard-cell application-specific integrated circuits (ASICs) in terms of logic density, circuit speed, and

power consumption for core logic are presented. [Online]. Available: https://dl.acm.org/doi/10.1145/1117201.1117205

[4] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends® in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008, tLDR: This survey reviews the historical development of programmable logic devices, the fundamental programming technologies that the programmability is built on, and then describes the basic understandings gleaned from research on architectures. [Online]. Available: http://www.nowpublishers.com/article/Details/EDA-005

[5] W. Huang, H. Wu, Q. Chen, C. Luo, S. Zeng, T. Li, and Y. Huang, "FPGA-Based High-Throughput CNN Hardware Accelerator With High Computing Resource Utilization Ratio," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 8, pp. 4069–4083, Aug. 2022, conference Name: IEEE Transactions on Neural Networks and Learning Systems TLDR: A novel composite hardware CNN accelerator architecture based on a row-level pipelined streaming strategy and an efficient data system with continuous data supply is designed to avoid the idle state of the CE. [Online]. Available: https://ieeexplore.ieee.org/document/9354493/?arnumber=9354493

[6] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, Jul. 2009, conference Name: IEEE Design & Test of Computers TLDR: The authors introduce the FSMD model, which forms the basis for synthesis, and discuss the main considerations in a high-level synthesis environment: the input description language, the internal representation, and the main synthesis tasks-allocation, scheduling, and binding. [Online]. Available: https://ieeexplore.ieee.org/document/5209958/?arnumber=5209958

[7] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS Today: Successes, Challenges, and Opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 1–42, Dec. 2022, tLDR: The progress of the deployment of HLS technology is assessed and the successes in several application domains are highlighted, including deep learning, video transcoding, graph processing, and genome sequencing. [Online]. Available: https://dl.acm.org/doi/10.1145/3530775

[8] V. Kathail, "Xilinx Vitis Unified Software Platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside CA USA: ACM, Feb. 2020, pp. 173–174, tLDR: This talk provides an overview of Vitis and Vitis AI development environments, which addresses the three major industry trends: the need for heterogenous computing, applications that span cloud to edge to end-point, and AI proliferation. [Online]. Available: https://dl.acm.org/doi/10.1145/3373087.3375887

[9] T. Kumar and K. Verma, "A Theory Based on Conversion of RGB image to Gray image," *International Journal of Computer Applications*, vol. 7, no. 2, pp. 5–12, Sep. 2010, tLDR: The use of color in image processing is motivated by two principal factors; first color is a powerful descriptor that often simplifies object identification and extraction from a scene, and second, human can discern thousands of color shades and intensities, compared to about only two dozen shades of gray. [Online]. Available: http://www.ijcaonline.org/volume7/number2/pxc3871493.pdf

[10] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, Sep. 1982, tLDR: Specific aspects of cache memories investigated include: the cache fetch algorithm (demand versus prefetch), the placement and replacement algorithms, line size, store-through versus copy-back updating of main memory, cold-start versus warm-start miss ratios, mulhcache consistency,

the effect of input /output through the cache, the behavior of split data/instruction caches, and cache size. [Online]. Available: https://dl.acm.org/doi/10.1145/356887.356892

[11] O. Vincent and O. Folorunso, "A Descriptive Algorithm for Sobel Image Edge Detection," 2009, tLDR: The Sobel operator performs a 2-D spatial gradient measurement on images to enhance the removal of redundant data, as a result, reduction of the amount of data is required to represent a digital image. [Online]. Available: https://www.informingscience.org/Publications/3351

Chapter 5

附录

5.1 附录 1：特别鸣谢

本项目基于的是 Xilinx University Progrm 中的 FPGA Sobel HLS 入门教程完成的，特别感谢开源社区对本项目的大力支持。（项目链接：`https://github.com/Xilinx/xup_high_level_synthesis_design_flow`）。

5.2 附录 2：优化后的 HLS 代码

```
130  void sobel(hls::stream<trans_pkt>& src, hls::stream<trans_pkt>& dst, int rows, int cols)
131  {
132      trans_pkt data_p;
133
134      PIXEL _sobel;
135
136      PIXEL LineBuffer[3][WIDTH];
137  #pragma HLS ARRAY_PARTITION variable=LineBuffer complete dim=1
138
139      PIXEL WindowBuffer[3][3] = {{0,0,0},{0,0,0},{0,0,0}};
140  #pragma HLS ARRAY_PARTITION variable=WindowBuffer complete dim=0
141
142      ap_uint<13> row, col;
143      ap_uint<2> lb_r_i;
144      ap_uint<2> top, mid, btm;//line buffer row index
145
146  // Loop initialing the row buffer:
147      for(col = 0; col < cols; col++)
148      {
149  #pragma HLS LOOP_TRIPCOUNT min=1 max=1280
150  #pragma HLS pipeline
151          LineBuffer[0][col] = 0;
152          data_p = src.read();
153          LineBuffer[1][col] = (PIXEL) data_p.data;
154      }
155
156      lb_r_i = 2;
157      for(row = 1; row < rows + 1; row++)
158      {
159  #pragma HLS LOOP_TRIPCOUNT min=1 max=720
160  // Rotate the relative order among LineBuffer
161          if(lb_r_i == 2)
162          {
163              top = 0; mid = 1; btm = 2;
164          }
165          else if(lb_r_i == 0)
166          {
167              top = 1; mid = 2; btm = 0;
168          }
169          else if(lb_r_i == 1)
170          {
171              top = 2; mid = 0; btm = 1;
172          }
173
```

图 5.1: 优化后的 HLS 代码

```
174             WindowBuffer[top][0] = WindowBuffer[top][1] = 0;
175             WindowBuffer[mid][0] = WindowBuffer[top][1] = 0;
176             WindowBuffer[btm][0] = WindowBuffer[top][1] = 0;
177
178     // Loop iterating over images:
179             for(col = 0; col < cols; col++)
180             {
181     #pragma HLS LOOP_TRIPCOUNT min=1 max=1280
182     #pragma HLS pipeline
183                 if(row < rows)
184                 {
185                     data_p = src.read();
186                     LineBuffer[btm][col] = (PIXEL) data_p.data;
187                 }
188                 else
189                     LineBuffer[btm][col] = 0;
190     // Update the WindowBuffer
191                 WindowBuffer[0][2] = LineBuffer[top][col];
192                 WindowBuffer[1][2] = LineBuffer[mid][col];
193                 WindowBuffer[2][2] = LineBuffer[btm][col];
194                 _sobel = sobel3x3_kernel(WB: WindowBuffer);
195                 WindowBuffer[0][0] = WindowBuffer[0][1];
196                 WindowBuffer[1][0] = WindowBuffer[1][1];
197                 WindowBuffer[2][0] = WindowBuffer[2][1];
198                 WindowBuffer[0][1] = WindowBuffer[0][2];
199                 WindowBuffer[1][1] = WindowBuffer[1][2];
200                 WindowBuffer[2][1] = WindowBuffer[2][2];
201
202                 if ((row == rows ) && (col == cols - 1))
203                     data_p.last = 1;
204                 else
205                     data_p.last= 0;
206                 data_p.data = _sobel;
207                 dst.write(din: data_p);
208             }
209             lb_r_i++;
210             if(lb_r_i == 3) lb_r_i = 0;
211         }
212     }
```

图 5.2: 优化后的 HLS 代码-2

5.3 附录 3：FPGA 安装 Jupyter 环境与课程项目地址

去官方的网站:https://www.pynq.io/boards.html找到对应自己板子的 img，本实验使用 PYNQ-z2 系列镜像，并使用 win32diskimager 烧录镜像（可能耗时较久，图 5.3、图 5.4）需要准备一个至少 8G 的 SD 卡（官方文档建议）。烧录完成后连接板卡，并按照官方文档访问板卡 Jupyter 服务完成登录，最后可以进入 Jupyter 界面（图 5.5）。具体可以参考官方教程https://www.youtube.com/watch?v=RiFbRf6gaK4&t=4s

本课程的 GitHub 仓库地址：https://github.com/nightt5879/FPGA

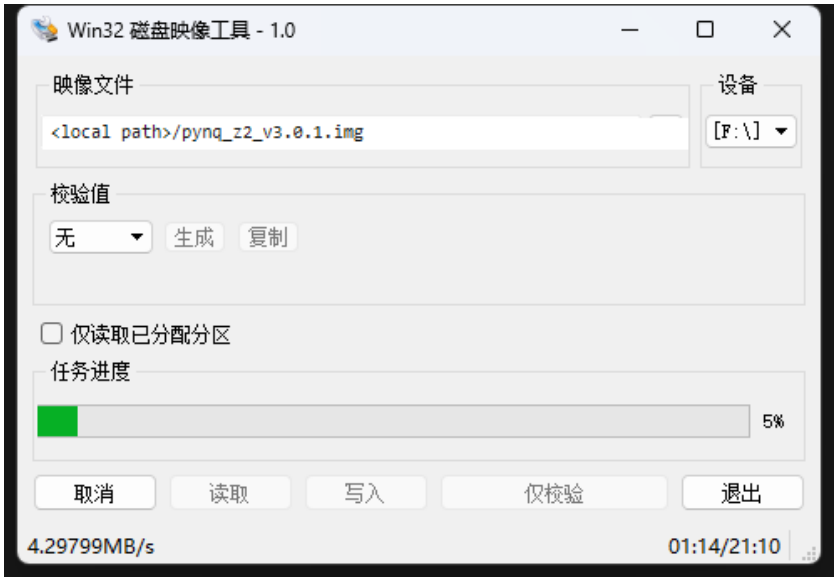


图 5.3: 烧录镜像

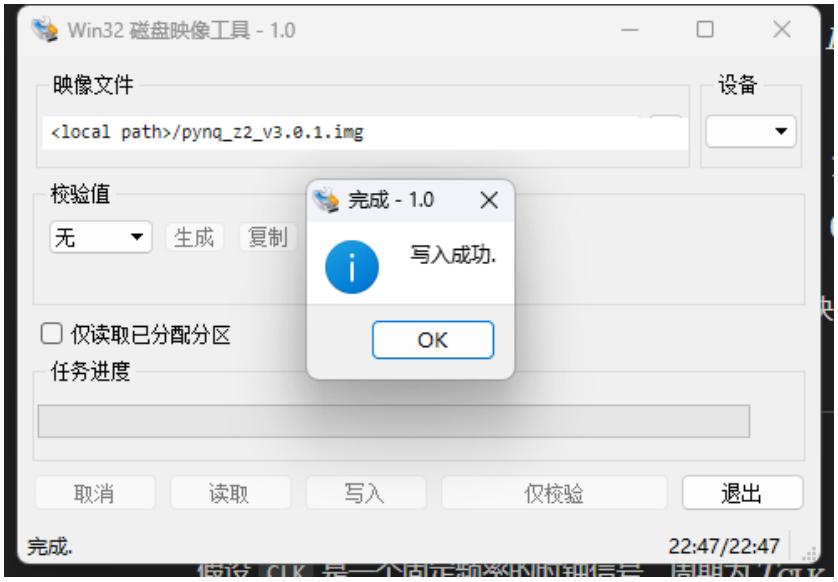


图 5.4: 烧录镜像-2

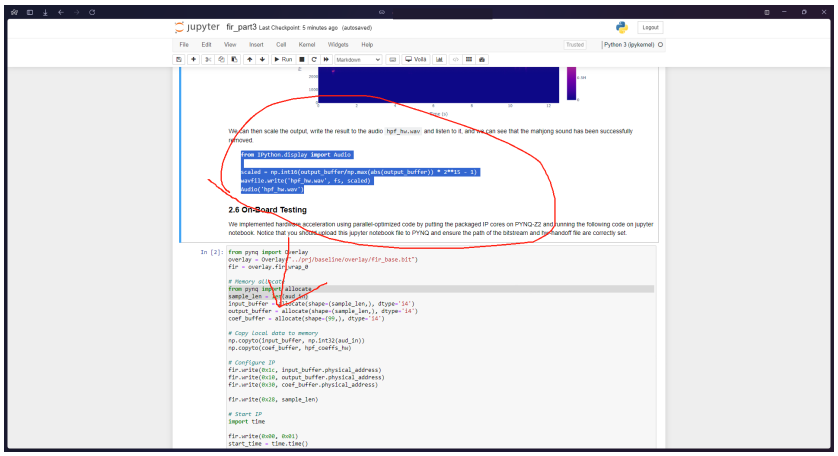
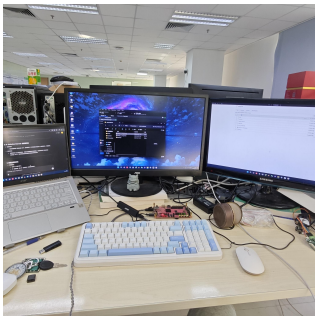
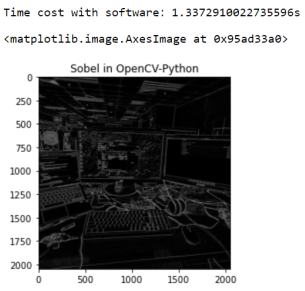


图 5.5: 进入 Jupyter 界面

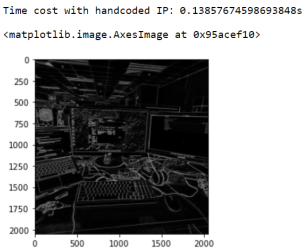
5.4 附录 4：测试结果



(a) 原始图像

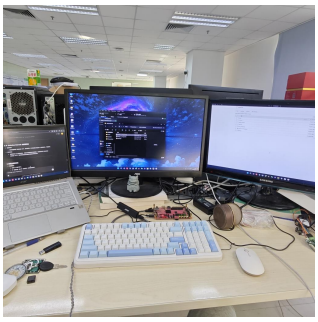


(b) Python 处理结果

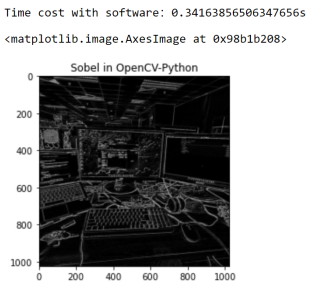


(c) IP 核处理结果

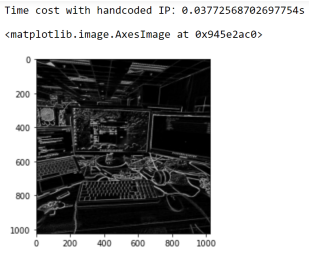
图 5.6: 分辨率为 2048 的图像处理结果



(a) 原始图像

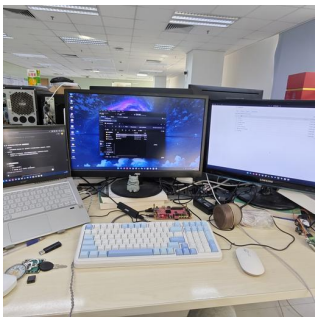


(b) Python 处理结果

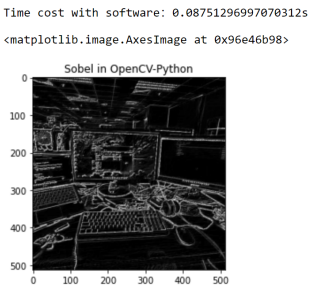


(c) IP 核处理结果

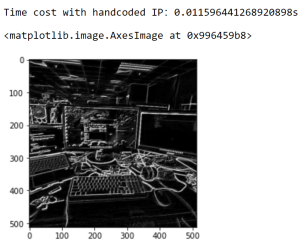
图 5.7: 分辨率为 1024 的图像处理结果



(a) 原始图像



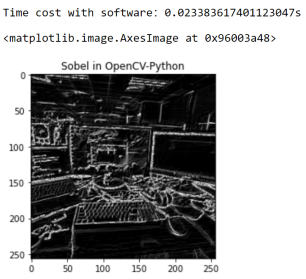
(b) Python 处理结果



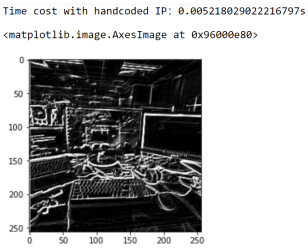
(c) IP 核处理结果

图 5.8: 分辨率为 512 的图像处理结果

(a) 原始图像



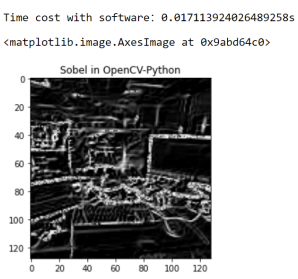
(b) Python 处理结果



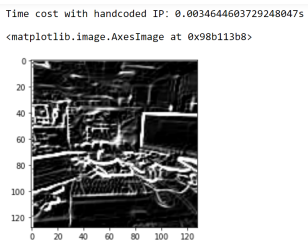
(c) IP 核处理结果

图 5.9: 分辨率为 256 的图像处理结果

(a) 原始图像



(b) Python 处理结果



(c) IP 核处理结果

图 5.10: 分辨率为 128 的图像处理结果

5.5 附录 5：图片出处

尊重创作者（即使可能也是转载的）是一件非常重要且严肃的事情，所以本次课程报告中会将所有非自己生成的图片注明出处！

图 2.1来自 GeeksforGeeks 平台：https://www.geeksforgeeks.org/matlab-rgb-image-representation/;图 2.2来自飞浆深度学习官方文档https://paddlepedia.readthedocs.io/en/latest/tutorials/CNN/convolution_operator/Convolution.html图 2.3、图 2.3、图 2.7、图 2.9、图 2.12、图 2.13、图 2.14、图 2.15来自 Xilinx University Progrmahttps://github.com/Xilinx/xup_high_level_synthesis_design_flow/blob/main/source/sobel/notebook